VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INFORMAČNÍCH SYSTÉMŮ
DEPARTMENT OF INFORMATION SYSTEMS

EVIDENCE SKLADOVÝCH ZÁSOB POMOCÍ QR KÓDŮ

EVIDENCE OF WAREHOUSE STOCKS USING QR CODES

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — **TOMÁŠ REBRO**
AUTHOR

VEDOUCÍ PRÁCE — **Ing. TOMÁŠ VOLF**
SUPERVISOR

BRNO 2019

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav informačních systémů (UIFS) Akademický rok 2018/2019

Zadání bakalářské práce

Student: **Rebro Tomáš**
Program: Informační technologie
Název: **Evidence skladových zásob pomocí QR kódů**
Evidence of Warehouse Stocks Using QR Codes
Kategorie: Web

Zadání:

1. Seznamte se s principy tvorby a zabezpečení webového API.
2. Analyzujte zabezpečení přístupu k fotoaparátu v prohlížeči na OS Android, zejména s ohledem na povolení přístupu pouze k definované stránce.
3. Po dohodě s vedoucím navrhněte kompletní webovou aplikaci (optimalizovanou zejména pro mobilní zařízení) nebo kombinaci mobilní a webové aplikace pro evidenci skladových zásob s maximálním důrazem na jednoduchost a intuitivnost pro uživatele. V případě mobilní aplikace se rovněž seznamte s principy tvorby mobilní aplikace.
   Současně navrhněte API, skrze nějž bude aplikace (včetně webové) zajišťovat požadavky.
   Uživatel bude mít možnost vytvořit mapu místnosti a polic; vytvořit jednotlivé položky; ke každé položce bude k vytištění vygenerován QR kód s popiskem; rovněž lze udělit přístup dalším osobám. Pomocí QR kódu bude konkrétní položka dále umisťována, přesouvána nebo odebírána z police, příp. zjišťováno info o položce. Dále bude možné prohledávat skladové zásoby (počty, umístění), nalézt umístění nejstarší/problémové položky skladových zásob nebo vybrat v aplikaci konkrétní položku a její nalezení zkontrolovat dle QR kódu a další dle dohody s vedoucím práce.
4. Navržené API a webovou/mobilní aplikaci implementujte (API a web v jazyce PHP 7), volbu použitých technologií zdůvodněte.
5. Ověřte funkčnost API a uživatelské aplikace.
6. Zhodnoťte dosažené výsledky, diskutujte další možné rozšiřování práce.

Literatura:

- Ahmad, H. W.: Building RESTful WebServices with PHP 7, 2017, ISBN: 978-1-78712-774-6.
- Mitchel, L. J.: PHP Web Services, 3rd edition, 2014, ISBN: 978-1-491-93309-1.

Podrobné závazné pokyny pro vypracování práce viz http://www.fit.vutbr.cz/info/szz/

Vedoucí práce: **Volf Tomáš, Ing.**
Vedoucí ústavu: Kolář Dušan, doc. Dr. Ing.
Datum zadání: 1. listopadu 2018
Datum odevzdání: 15. května 2019
Datum schválení: 14. ledna 2019

Abstrakt

Cieľom tejto práce je navrhnúť a vytvoriť webovú aplikáciu pre evidenciu skladových zásob. Aplikácia je optimalizovaná najmä pre mobilné zariadenia. Aplikácia vytvára intuitívne, graficky prívetivé prostredie pre uživateľa, jednoduché vyhľadávanie produktov a ich identifikáciu na základe QR kódov. Vytvorené riešenie umožňuje definovanie vlastných kategórií produktov, ich atribútov, umiestňovanie produktov cez mapu skladu a zobrazenie súhrnných informácií o sklade.

Abstract

The aim of this thesis is to design and create a web application for evidence of warehouse stocks. The application is optimized especially for mobile devices. The application creates an intuitive, graphically friendly user experience, easy warehouse stock search and QR code identification of stocks. The solution allows you to define your own stock categories, their attributes, stock placement through the warehouse map and view warehouse summary information.

Kľúčové slová

PHP, Laravel, Eloquent, MySQL, ReactJS, Redux, Material Design, Informačný systém, evidencia skladu, QR kód

Keywords

PHP, Laravel, Eloquent, MySQL, ReactJS, Redux, Material Design, Information system, evidence of warehouse, QR code

Citácia

REBRO, Tomáš. *Evidence skladových zásob pomocí QR kódů*. Brno, 2019. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Tomáš Volf

Evidence skladových zásob pomocí QR kódů

Prehlásenie

Prehlasujem, že som túto bakalársku prácu vypracoval samostatne pod vedením pána Ing. Tomáša Volfa. Uviedol som všetky literárne pramene a publikácie, z ktorých som čerpal.

.....
Tomáš Rebro
14. mája 2019

Poďakovanie

Chcel by som poďakovať vedúcemu práce za to, že mi umožnil pracovať na tejto bakalárskej práci a za jeho poskytnutú odbornú pomoc.

Obsah

Kapitola 1

Úvod

Tento text hovorí o tvorbe webovej aplikácie, určenej na evidenciu skladových zásob pomocou QR kódov. Aplikácia je primárne určená pre vedúceho práce na evidenciu zásob ovocného sadu, ale je navrhnutá spôsobom, vďaka ktorému si môže nájsť uplatnenie aj v skladoch s iným zameraním. Práca je rozdelená na teoretickú a praktickú časť, kde sa každá z týchto častí samostatne venuje serverovej a klientskej zložke.

Prvá teoretická kapitola sa zaoberá princípom tvorby serverovej časti. Podrobnejšie je tam rozobraná REST architektúra a spomenuté sú aj jej alternatívy. Vysvetlená je potreba frameworku pri tvorbe takejto aplikácie a popísane sú tie najpoužívanejšie z nich.

Ďalšia kapitola hovorí o vývoji webového uživateľského rozhrania, kde je porovnanie populárnych frontendových frameworkov. Ukázané su možnosti prístupu ku kamere zariadenia vzhľadom na podporu webových prehliadačov.

Návrh riešenia sa samostatne venuje návrhu databázovej, aplikačnej a prezentačnej vrstve. Pri databázovej vrstve je časť venovaná návrhu EAV (entity attribute value) modelu a návrhu uživateľských prístupov. V návrhu aplikačnej vrstvy je popísanie prípadov použitia jednotlivých HTTP metód, využitý princíp na optimalizáciu počtu požiadaviek a opísaná je aj autorizácia endpointov. V prezentačnej časti je spomenutý výber frameworku a popísaný je aj návrh jednotlivých stránok aplikácie.

Implementačná časť sa samostatne venuje implementácií REST API, štruktúre aplikácie a akým spôsobom sú implementované niektoré jej časti. V uživateľskom rozhraní je popísaná štruktúra aplikácie hlavne ohľadom na využitie knižnice `Redux` a popísanie niektorých zaujímavých komponentov.

Kapitola 2

Princípy tvorby webovej služby

API (application programming interface) je rozhranie, cez ktoré sa softvér reprezentuje pre ostatné programy, pre ľudí a v prípade webového API, do sveta prostredníctvom internetu. Aj ked je jeho dizajn vytvorený pre prácu s inými softvérmi, často je zamýšlané, aby boli zrozumiteľné a použité ľudmi, ktorí pracujú na týchto iných softvéroch. O webových API môžme hovoriť, ako o stavebných blokoch umožňujúcich interoperabilitu pre väčšinu platforiem na webe. Pre spoločnosti s víziou vytvorenia platformy, ktorá je prístupná pre každého, sú aplikačné rozhrania veľmi dôležitou časťou a predstavujú kľúčové komponenty spoločností ako Amazon, Stripe, Google a Facebook.

2.1 Architektúra orientovaná na službu

SOA (service oriented architecture) je architektonický štýl webových služieb definujúce určité štandardy a stanovuje najlepšie spôsoby, ako dizajnovať a programovat webové služby. SOA je nezávislá a poskytuje pokyny na kombinovanie služby s inými službami.

2.1.1 Zásady návrhu

Architektúru orientovanú na službu aplikáciou navrhnutou podľa nasledujúcich 8 zásad:

- Štandardizovaný kontrakt služby - služby vyjadrujú svoj účel pomocu kontraktu. Toto je pravdepodobne najzásadnejšia zásada, ktorá podstatne prikazuje potrebu architektonického riešenia byť rozdelená na časti a distribuovaná štandardizovaným spôsobom.
- Služba bez viazanosti - Viazanosť v tomto prípade ukazuje na závislosť medzi dvomi vecami. Táto zásada zriaduje špecifický typ vzťahu vo vnútri a mimo hranice služby s dôrazom na redukovanie závislosti medzi kontraktom služby, jej implementáciou a jej konzumermi. Propaguje nezávislý návrh a evolúciu logiky služby, zatiaľ čo garantuje interoperabilitu.
- Abstrakcia služby - abstrakcia je v mnoha aspektoch architektúry. Kontraky služby obsahujú iba nevyhnutné informácie a informácie o službe sú limitované na to, čo je publikované v kontraktoch. Základ je, že tento princíp hovorí o potrebe skryť tak veľa podrobností o službe, ako je možné. Toto nám priamo umožňuje predošle opísanú zásadu o strate viazanosti.
- Znovupoužitie služby - Keď vytvárame službu, snažíme sa, aby bola schopná a užitočná pre viac ako len jeden účel.

- Autonómnosť služby - Pre služby vykonávajúce svoju schopnosť konzistentne a spoľahlivo, ich riešenie vyžaduje mať signifikantnú úroveň kontroly nad jej prostredím a zdrojmi.
- Bezstavovosť služby - Manažment nadmerných informácií o stave môže kompromitovať dostupnoť služby rovnako, ako odhad jej správania. Služby sú preto navrhnuté na zachovanie stavu iba v prípade, keď je to nutné. Toto je ďalšia zásada týkajúca sa viac navrhnutia logiky samotnej aplikácie, ako jej kontraktu.
- Ifentifikovatelnosť služby - Služby sú doplnené metadátami, podľa ktorých môžu byť efektívne rozpoznané a interpretované.
- Princíp skladania - Týmto princípom je zaistené možné stretnutie jednotlivých služieb a zaistenie ich vzájomnej synergie. Ide o skladanie nezávislých komponentov (služieb) do výsledného systému tak, aby dokázali zvládnut komplexnejší problém.

2.2 Zdrojovo orientovaná architektúra

Nápad ROA[14] (zdrojovo orientovaná architektúra) je použitie jednoduchých, ľahko pochopiteľných a známych webových technológií (HTTP, URI a XML) spolu so základnými návrhovými princípmi

Primárne zameranie webových služieb je spájanie informačných systémov a ROA definuje štrukturálny dizajn, alebo množinu pravidiel pre podporu a implementáciu interakcií v akomkoľvek pripojenom zdroji. Ktorákoľvek entita biznisu môže byť reprezentovaná ako zdroj a môže byť dostupný cez URI.

Napríklad, v organizačnom systéme pre ľudské zdroje každý zamestnanec je entita a plat, zamestnancove detaily sú opis tejto entity.

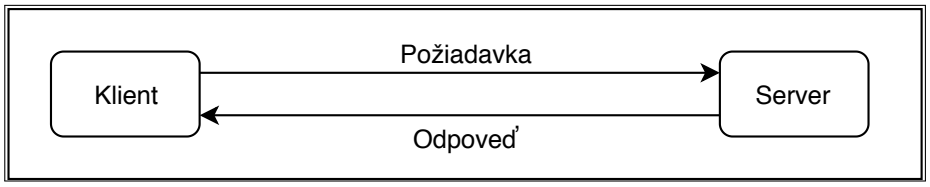
Jednoduché porovnanie pre objektovo orientované a zdrojovo orientovaný koncepty je v tabuľke 2.1, ktorá ponúka náhľad, čo ROA znamená.

Tabuľka 2.1: Porovnanie objektovo orientovanej a zdrojovo orientovanej architektúry

Objekty v objektovo orientovanej architektúre	**Zdroje v ROA**
Každá entita je definovaná ako objekt	Entity sú služby
Objekt má atribúty a akcie	Služba má popisy a kontrakt
Objecty musia udržiavať stav na interakciu	Interakcie po sieti s definovanou lokáciou a adresou

Benefity zdrojovo orientovanej architektúry:

- Nezávislosť na klientských kontraktoch: oslobodené od kontraktných formulácií, pretože celý web je založený na HTTP operáciach
- Explicitný stav: samotný zdroj reprezentuje stavy, servery neobdržiavajú neznáme, aplikáciou špecifické požiadavky. Server nemusí udržiavať reláciu klienta, ktorý odoslal požiadavku a rovnako klient nepotrebuje vedieť, s ktorým serverom komunikoval.



Obr. 2.1: Klient server komunikácia.

- Škálovateľnosť a výkon: Škálovateľnost s ROA je daná jej charakteristikami, ako je explitcitný stav, žiadne obmedzenie spôsobené kontraktmi a upustenie od naviazania klienta na server (relácie). Zlepšenie výkonu ohľadom na dobu odozvy pre ROA využitie medzipamäte, vyvažovanie záťaže, indexovanie a vyhľadávanie hrá dôležitú rolu v lepšom výkone.

2.3 REST

REST (Representational State Transfer) je architektonický štýl pre sieťovú aplikáciu, na ktorom bol založený paralelne vyvíjaný HTTP 1.1. Ako RESTful označujeme systém alebo webovú službu, ktorá dodržiava nasledujúcich šest podmienok. V opačnom prípade služba nemože byť považovaná za RESTful.

2.3.1 Klient - server

REST separuje klienta a server. Táto podmienka vznikla kvôli separácií záujmov. Znamená to, že server a klient majú rozdielne úlohy a žiadny z nich nie je zodpovedný za funkciu toho druhého. Napríklad, klient nie je zodpovedný za ukladanie dát na serveri, pretože je to zodpovednosť serveru. Obidve časti, server a klient, vykonávajú svoju vlastnú činnosť a plnia ich vlastné úlohy, čo robí ich prácu jednoduchšou. V praxi to znamená, že server môže byť lepšie škálovateľný a klient nezávislý a viac interaktívny.

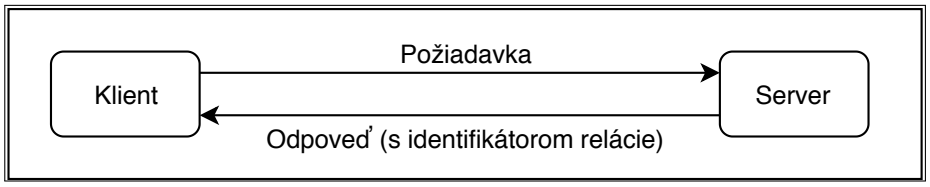
2.3.2 Bezstavovosť

Komunikácia medzi klientom a serverom je bezstavová. Každá požiadavka od klienta musí obsahovať všetky informácie potrebné k jej vybaveniu. Znamená to, že v komunikácií neexistuje žiadny stav iný, ako obsah požiadavky. Odpoveď, ktorú klient obdrží, je vybavená na základe požiadavky, bez ovplyvnenia akýchkoľvek iných stavov, ktoré nie sú jej súčasťou. To znamena, že pokiaľ parametre požiadaviek zostanú nezmenené, rovnaké požiadavky nikdy nevyústia k rôznym odpovediam.

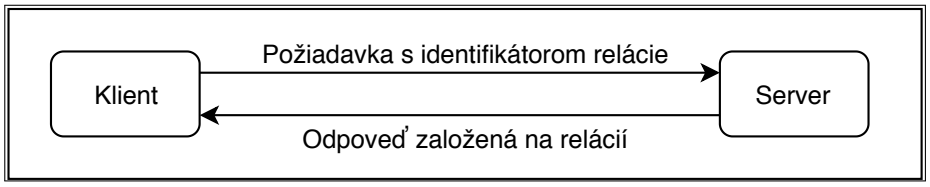
Ak je potrebné vytvoriť reláciu (zachovať stav), môže byť vytvorená na základe tokenu alebo iného identifikátoru, ktorý je súčasťou požiadavky. Tento identifikátor umožní zoskupenie množiny požiadaviek na zachovanie stavu. Možeme si to ukázať na príklade web požiadavky, kde HTTP komunikácia nie je nič viac ako požiadavka poslaná zo strany klienta na server a odpoveď poslaná späť ku klientovi zo servera, ako ukazuje obr. 2.1.

Ak potrebujeme vytvoriť reláciu, informácie o nej sú uložené na serveri. Identifikátor relácie bude v rámci odpovede poslaný klientovi. Nasledujúce požiadavky od klienta budú obsahovať identifikátor relácie, podľa ktorého bude server schopný identifikovat klienta a načítať data spojené s danou reláciou. Komunikácia je znáznornená na obrázku 2.2.

Každá nasledujúca požiadavka bude vyzerať ako to vidíme na Obrázku 2.3.



Obr. 2.2: Odpoveď s identifikátorom relácie.



Obr. 2.3: Využitie relácie zo strany klienta.

2.3.3 Možnosť využitia medzipamäte

Odpoveď RESTful webovej služby musí definovať či je pre ňu možné využiť medzipamäť alebo nie a podľa tejto informácie sa klient rozhodne či si odpoveď uložiť. Vo výsledku to prinesie lepší výkon, pretože klient pri ďalšej rovnakej požiadavke už nebude komunikovať so serverom, ale použije data zachované v medzipamäti.

2.3.4 Jednotné rozhranie

Táto podmienka je najvýraznejšia. Oddeľuje rozhranie od implementácie, ako v každom dobre navrhnutom systéme. Jednotné rozhranie má štyri vlastnosti.

Identifikácia zdroja

Zdroj bude identifikovaný v požiadavke. Napríklad, vo webovom REST systéme to bude identifikované v URI. Odpoveď nie je ovplyvnená tým, ako je zdroj na serveri uložený.

Manipulácia zdrojov cez reprezentácie

Vlastnosť hovorí, že klient by mal mať reprezentáciu zdroja, v ktorej má dostatok informácií na jeho modifikáciu alebo zmazanie. Napríklad, vo webovej REST službe môže byť každá operácia nad zdrojom vykonaná použitím HTTP metódy a URI.

Samopopisujúce správy

Každá správa obsahuje dostatok informácií, popisujúcich, ako túto správu spracovať. Môže to byť popis formátu správy (JSON, XML).

HATEOAS (Hypermédiá ako aplikačný stav)

Jeden z princípov REST architektúry, ktorým sa odlišuje od ostatných sieťových architektúr. Znamená to, že server vo svojej odpovedi uvedie, aké ďalšie akcie sú možné a ako ich vykonať. V ideálnom REST rozhraní by klient poznal iba hlavnú URI a postupným posielaním požiadaviek, môže klient využiť kompletné ponúkané rozhranie. V praxi sa to

používa napríklad na dynamické presmerovanie na iný server, bez akýchkoľvek zmien na strane klienta.

2.3.5 Kód na požiadanie

Znamená to, že server môže pridať funkcionalitu REST klientovi poslaním kódu spustiteľným klientom. V kontexte webovej služby to môže byť časť JavaScript kódu, ktorý odošle server do prehliadača. API môže byť označované za RESTful aj bez posktovania kódu na požiadanie.

2.3.6 Vrstvený systém

REST systém môže mať viacero vrstiev a ak klient požaduje odpoveď, tá nemôže byť rozlíšiteľná či pochádza zo servera alebo z iného middle-ware servera. Jedna serverová vrstva môže byť nahradená inou, ak to neovplyvní rozhranie zo strany klienta.

2.4 Alternatívy k REST

2.4.1 SOAP

Protokol na výmenu správ založených na XML prostredníctvom siete. Jeho účelom je zabezpečiť rozšíriteľnosť, neutralitu a nezávislosť. Môže operovať na protokoloch ako sú HTTP, SMTP, TCP, UDP alebo JMS. Protokol je založený na XML pozostávajúci z nasledujúcich častí:

1. Koreňový element Envelope, ktorý definuje XML dokument, ako SOAP správu.
2. Telo správy (Body element) obsahujúca informácie o volaniach a odpovediach.
3. Chybová časť (Fault element) obsahujúca chyby a status informácie.

SOAP vs. REST porovnanie

Aj ked je dnes REST veľmi populárny, SOAP ma stále svoje uplatnenie vo svete webových služieb. Je to voľba typická najmä v starších systémoch, ale má určité výhody v porovnaní s REST:

- Nezávislé na jazyku, platforme a transportnej vrstve (REST využíva iba HTTP).
- Dobrá funkčnosť v distribuovaných prostrediach (REST predpokladá priamu komunikáciu point-to-point).
- Štandardizovaný.
- Poskytuje významný rozšíriteľnosť vo forme WS* štandardov.
- Zabudované spracovanie chýb.

Pre väčšinu služieb je jednoduchšie a flexibilnejsie použiť REST. V porovnaní so SOAP ma tieto výhody:

- Žiadne drahé nástroje nie sú potrebné na interakciu s webovou službou.

- Rýchlejšia krivka učenia.
- Efektívnosť (SOAP používa XML pre všetky správy, REST môže použiť efektívnejší formát).
- Rýchlosť (žiadne rozsiahle spracovávanie).
- Technologicky bližší k ostatným technológiam používaných vo webových službách.

2.4.2 GraphQL

Dotazovací jazyk, ktorý je vo veľa zdrojoch nesprávne označovaný za nástupcu REST. Ponúka alternatívu ku REST architektúre. Najdôležitejšia prednosť GraphQL je možnosť načítania rôznych zdrojov pomocou jednej požiadavky a tým pádom uľahčenie práce zo strany klienta [11].

GraphQL vs. REST porovnanie

Výhody GraphQL:

- Možnosť požiadať presne o tie data, ktoré práve potrebujeme v pohľade, v jednej požiadavke.
- Klient používa dotazovací jazyk, ktorý eliminuje potrebu mať definovaný formát dát na strane servera.

Nevýhody GraphQL:

- Nie je možné použiť HTTP medzipamäť.
- Bez podpory verzovania. Neexistuje žiadny jasný indikátor, ktorý by hovoril, že atribút alebo hodnota je zastaralá.

2.4.3 gRPC

Technológia vytvorená spoločnosťou Google, ako open source evolúcia ich internej RPC (remote procedure calls) technológie. Architekrúra gRPC je postavená na niekoľkých vrstvách [7]. Najnižšia z nich je transportná, pužívajúca HTTP/2, ponúkajúca rovnakú sémantiku ako HTTP1.1, s dôrazom na vyššiu efektivitu a lepšiu bezpečnosť. Ďalšia vrstva je kanál, abstrakcia nad transportnou vrstvou, definujúca volacie konvencie a implementujúca mapovanie RPC na transportnú vrstvu. Volanie pozostáva z klientom poslaného mena služby a názov metódy, prípadne metadáta požiadavky. Na tejto vrstve je správa iba sekvenciou bajtov. Na poslednej vrstve sú definované podmienky rozhrania a dátové typy.

Jedna z kľúčových komponentov gRPC je technológia zvaná Protocol Buffers (serializácia štrukturovaných dát). Z pohľadu webových aplikácií ponúka Javascriptovú knižnicu `gRPC-Web`, pomocou ktorej dostane prehliadač prístup ku gRPC službe. Momentálne je možné použiť túto knižnicu iba pre prístup k backendu napísaného v jazyku Javascript.

2.5 PHP frameworky

Termín framework sa vzťahuje na knižnice súborov, ktoré obsahujú niekoľko základných funkcií. Cieľom frameworku je poskytnúť základ, ktorý môžeme využiť pre efektívnejší vývoj. Preto v sebe zahŕňa mnoho funkcií, ktoré by sme v opačnom prípade museli od základov implementovať. Je jednoduché vidieť, prečo je tak benefitné použiť individuálne komponenty, alebo balíčky prístupné PHP programátorom. S balíčkami je niekto iný zodpovedný za vývoj a údržbu izolovanej časti kódu, ktorá má dobre definovanú funkciu a teoreticky tento človek hlbšie rozumie jednotlivej komponente, ako niekto iný. Frameworky ako Laravel, Symfony, Lumen a Slim balia kolekciu komponentov tretích strán do seba vlastným riešením, pomocou konfiguračných súborov, poskytovateľov služby, predpísanou adresárovou štruktúrou a aplikačný bootstrap. Benefit použitia frameworku je vo všeobecnosti, že niekto iný urobil rozhodnutia nie iba o jednotlivých komponentoch pre nás, ale taktiež ako tieto komponenty spolu pracujú.

Povedzme, že si vytvárame novú webovú aplikáciu bez benefitov frameworku. Kde začneme? Pravdepodobne by to malo vedieť smerovať HTTP požiadavky, takže musíme zhodnotiť všetky dostupné knižnice na HTTP požiadavky a odpovede a vybrať jednu z nich. Potom musíme vybrať smerovač a pravdepodobne budeme musieť vytvoriť aj nejakú formu konfiguračného súboru pre endpointy. Akú syntax na to použijeme? Kde tento súbor umiestnime? K týmto endpointom budeme potrebovať obslužný kód (controller). Kde sa budú nachádzať a ako sa načítajú? Pravdepodobne budeme potrebovať kontajner na vkladanie závislostí na kontroléry a ich závislosti, ale ktorý?

Frameworky riešia tento problém poskytovaním pozorne zváženým odpovediam na tieto otázky so zárukou, že jednotlivé vybrané komponenty správne spolu pracujú. Rovnako poskytujú konvencie, ktoré redukujú veľkosť kódu, ktorému musí nový programátor v tíme porozumieť. Ak rozumie, ako funguje smerovanie na jednom projekte s rovnakým frameworkom, bude rozumieť, fungovanie vo všetkých ostatných projektoch.

Keď hovoríme o PHP frameworkoch, máme na výber veľa možností. Neexistuje žiadny ideálny framework, všetky sú unikátne a majú svoje klady a zápory.

2.5.1 Krátka história PHP frameworkov

Ruby on Rails

David Heinemeier Hansson vydal prvú verziu Ruby on Rails v roku 2004 a je ťažké nájst webový aplikačný framework, ktorý by ním odvtedy nebol ovplyvnený určitým smerom [13].

Rails popularizoval MVC (Model View Controller), RESTful JSON aplikačné rozhrania, konvencie ohľadom konfigurácií, ActiveRecord a veľa ďalších nástrojov a konvencí, ktoré silne ovplyvnili smer, ktorým webový vývojári tvorili ich aplikácie.

Prítok PHP frameworkov

Pre väčšinu vývojárov bolo zreteľné, že Rails a podobné webovo aplikačné frameworky boli vlna budúcnosti a PHP frameworky, vrátane tých nesporne imitujúcich Rails, začali rýchlo prenikať.

CakePHP bol prvý v roku 2005, čoskoro nasledovaný Symfony, CodeIgniter, Zend Framework a Kohana (CodeIgniter fork). Yii prišlo v roku 2008, Aura a Slim v 2010. 2011

priniesol FuelPHP a Laravel, obidve neboli odnože CodeIgniter, ale namiesto toho boli považované za alternatívy.

Niektoré z týchto frameworkov boli viac podobné Rails, zameriavajúce sa na databázové objektové relačné mapovanie (ORM), MVC štruktúru a iné nástroje so zameraním na rýchly vývoj. Iné, ako Symfony a Zend, zamerané viac na biznisové návrhové vzory a elektronický obchod.

Preto si predstavíme tie najpoužívanejšie z nich podľa [15].

Laravel
Open source webový framework určený pre vývoj webových aplikácií, používajúci MVC (model-view-controller) architektúru. Tým vývojára núti oddeliť svoju aplikáciu na dátový model, uživateľské rozhranie a riadiacu logiku do troch nezávislých komponentov. Je to jeden z najpoužívanejších frameworkov, čo v praxi znamená, že je k dispozícií mnoho zdrojov, návodov a diskusií s témami týkajúcich sa práve tohto frameworku. Laravel využíva Composer ako správcu závislostí. S databázou je možné pracovať pomocou Eloquent ORM, čo prináša interné metódy pre definovanie vzťahov medzi databázovými objektami. Podľa architektonického návrhového vzoru Active Record sú databázové tabuľky prezentované ako triedy s inštanciami naviazanými na jednotlivé záznamy v tabuľke.

Symfony
Tento framework ponúka vývojárom znovupoužiteľné komponenty a vstavanú testovaciu funkcionalitu. Učiaca krivka je dosť strmá a framework nie je jednoduchý. Ľudia bez predchádzajúcich skúseností s týmto frameworkom budú pri začiatkoch potrebovať čas a úsilie. Vývojár má plnú kontrolu nad konfiguráciou od adresárovej štruktúry až po cudzie knižnice. Takmer všetko môže byť prispôsobené. Používa návrhový vzor MVC [12].

Zend Framework
Na správu balíčkov používa Composer rovnako ako Laravel. PHPUnit pre testovanie, Travis CI pre priebežnú integráciu. Rovnako ponúka podporu MVC v kombinácií s Front Controller. Čo sa týka adresárovej štruktúry, tak tento framework nepredpisuje žiadnu konkrétnu aplikačnú štruktúru.

Kapitola 3

Princípy vývoja webového uživateľského rozhrania

Tvorba uživateľského rozhrania so samotným Javascriptom je možná a týmto štýlom sa vytvárali web stránky veľa rokov. V priebehu rokov, niekoľko knižníc zaistilo začlenenie rôznych prvkov a možnosť znovupoužiteľnosti kódu. Ako rástli, niektoré z týchto knižníc sa spojili dokopy, aby vytvorili framework pomáhajúci vývojárom vytvoriť kompletné web stránky od začiatku do konca, zatiaľ čo poskytujú odpovede v rôznych aspektoch vyžadujúc modernými web stránkami, ako je smerovanie, autentifikácia, manažment stavov a tak ďalej.

3.1 Porovnanie frontend frameworkov

Mnoho webových aplikácií je dnes založených z väčšej časti na prezentačnej vrstve. Zapríčinil to najmä stále populárnejší JavaScript a jeho sila v tvorbe interaktívnych uživateľských rozhraní. Jeden z najpopulárnejších jazykov poskytuje taktiež množstvo frameworkov a knižníc. Medzi nimi sú najrozšírenejšie frameworky Angular, ReactJS a VueJS. Zjednodušené porovnanie podľa[1] možeme vidieť v tabuľke 3.1).

3.1.1 Angular

Angular je JavaScript MVVM (Model-View-ViewModel) framework, vytvorený v roku 2009, určený pre tvorbu interaktívnych web aplikácií. Stavebné bloky Angularu sú moduly (NgModules), ktoré vytvárajú kompilačný kontext pre komponenty. Moduly skladajú súvisiaci kód do funkčných celkov. Angular aplikácia je definovaná súborom modulov. Samotné komponenty definujú pohľady, ktoré vytvárajú súbor prezenčných elementov vybraných a modifikovaných na základe programovej logiky a dát. Komponenty využívajú služby (services), špecifikujúce funkcionality, ktorá nie je priamo spojená s pohľadmi. Služby môžu byť vkladané do komponentov ako závislosti, čo vytvára modulárny, znovupoužiteľný a efektívny kód. Komponenty aj služby sú triedy s dekorátormi, definujúcimi ich typ a metadáta, podľa ktorých ich Angular používa.

Výhody Angularu

- Vytvorený paralelne s Typescriptom, pre ktorý má výnimočnú podporu.
- MVVM (Model-View-ViewModel) umožňuje vývojárom pracovať oddelene na rovnakej časti aplikácie s použitím rovnakých dát.

Tabuľka 3.1: Tabuľka porovnania najpoužívanejších frameworkov.

Parameter alebo atribút	**Angular**	**React**	**Vue**
Podpora	Google	Facebook	Komunita
Prevládajúca architektúra	MVC	Flux	Flux
Architektonická flexibilita	Nie	Áno	Áno
CLI	angular-cli	Žiadne oficiálne CLI, ale bežne sa používa https://github.com/facebook/create-react-app	Vue-cli dovoluje založiť projekt s použitím chcenej šablóny ako webpack alebo browserify. Avšak keď je už projekt založený, nemá ďalšie využitie
Dokumentácia	Jednoduchá dokumentácia dostupná	Oficiálna dokumentácia je limitovaná a pre začiatočníka nemusí byť najvhodnejšia	Odporúčaná práca s dokumentáciou
Znovupoužitie kódu	Áno	Nie, iba CSS	Áno, CSS a HTML
Čo majú vývojári na ňom najradšej	Jeden z najstarších frameworkov s modularitou	Node Tree a Virtual DOM	Určitá kombinácia Angularu a ReactJS
Známe použitie	Google, Forbes, weather.com	PayPal, Uber, Netflix	Expedia, Alibaba, Nintendo

- Štruktúra a architektúra špecificky vytvorená pre škálovateľnosť projektu.
- Detailná dokumentácia poskytujúca všetko potrebné pre vývoj.
- Vkladanie závislostí je dôležitý aplikačný návrhový vzor, ktorý zvyšuje efektívnosť a modularitu aplikácie.
- Niekoľko zabudovaných schopností, ako je validácia formulárov a HTTP klient.
- Navrhnutý na testovateľnosť

Nevýhody Angularu

- Ťažké použitie spolu s inými jazykmi mimo TypeScript.
- Nekompatibilita so staršími prehliadačmi.

3.1.2 React

JavaScriptová knižnica[10] vhodná na tvorbu single-page alebo mobilných aplikácií. Pre komplexnejšie aplikácie je zvyčajne potreba ďalších knižníc pre manažment stavu, smerovanie a interakciu s API.

Výhody Reactu

- Ľahký na učenie vďaka jednoduchému dizajnu, použitia JSX (syntax podobná HTML) pre šablóny a detailnú dokumentáciu.
- Vývojár používa väčšinu času moderný JavaScript a menej času s framework špecifickým kódom.
- Veľmi rýchly kvôli React Virtual DOM implementácií a rôznym rendrovacím optimalizáciam.
- Podpora pre rendrovanie na strane servera, čo z neho robí silný framework pre obsahovo orientované aplikácie.
- Jednosmerný tok dát, čo v praxi znamená, menej nechcených vedľajších účinkov.
- Aplikácie môžu byť typovo bezpečné s Microsoft TypeScript alebo Facebook Flow (obidve majú natívnu podporu pre JSX).
- Migrácia medzi verziami je vo všeobecnosti veľmi jednoduchá.

Nevýhody Reactu

- Spôsob vývoja necháva na samotných vývojárov. Vo viacčlenných týmoch je toto možné riešiť dobrým vedeným projektu a dobrými procesmi.
- Komunita je rozdelená na dva tábory. Jeden si myslí, že je najlepšie oddeliť CSS moduly a druhý je za vkladanie CSS priamo do JavaScriptu.
- Miešanie šablón s logikou.

3.1.3 Vue.js

Jeho autori ho nazývajú progresívny framework[9]. Pretože umožňuje začať vytvárať aplikáciu s minimálnym úsilím, kedže Vue.js knižnica je zameraná iba na prezentačnú vrstvu. Neskôr, ked požiadavky narastú, môžme adaptovať Ďalšie knižnice.

Výhody Vue.js

- Detailná dokumentácia, ktorá veľmi pomôže vývojárovi, ktorý má iba základné znalosti HTML a JavaScriptu.
- Veľmi jednoduchý prechod z Angularu alebo Reactu, vdaka podobnosti z hľadiska dizajnu a architektúry.
- Môže byť použitý pre tvorbu single-page ako aj pre komplexnejsie webové rozhrania aplikácií. Menšie časti môžu byť jednoducho integrované do existujúcej infraštruktúry bez negatívnych efektov na celý systém.
- Malá veľkost (okolo 20 kB), čo pomáha rýchlosti a dosiahnutiu lepšieho výkonu v porovnaní s inými frameworkami.

Nevýhody Vue.js

- Málo zdrojov v porovnaní s Reactom a Angularom.
- Mierne riskovanie pri integrovaní do väčších projektov, kde môžu nastať problémy a neexistujú skúsenosti s ich riešením.

3.1.4 Redux

Pri vytváraní väčšej aplikácie, sa dostaneme do stavu, kedy je práca s aplikačným stavom čoraz náročnejšia. Aplikačný stav zahŕňa odpovede zo serverov, medzipamäť a data, ktoré ešte neboli odoslané na server. Avšak, stav uživateľského rozhrania konštantne naberá na komplexite. Napríklad smerovanie je často implementované na strane klienta, takže nie je treba obnoviť a znovunačítať celú aplikáciu pri načítaní novej stránky. Smerovanie na strane klienta má pozitívny vplyv na výkon aplikácie, ale znamená to, že klient sa musí postarať o ešte viac stavov. Správa všetkých stavov môže byť náročná, ak to nerobíme správne, môže sa aplikačný stav rýchlo vymknút z pod kontroly. Je to ťažké, pretože často miešame dva koncepty, ktoré môžu byť nepredvídateľné, keď ich spojíme dokopy: asynchronicita a mutácia[3].

Asynchronicita znamená, že zmeny sa môžu udiať kedykoľvek. Napriklad, uživateľ stlačí tlačidlo, ktoré spôsobí vytvorenie požiadavky na server. Nevieme, kedy sa vráti odpoveď zo servera a kvôli výkonu aplikácie, na ňu ani nechceme čakať. Preto používame asynchronicitu, kedy začneme reagovat na odpoveď až vtedy, keď pride. Ale nedokážeme predvídať, kedy sa to stane.

Mutácia reprezentuje každú zmenu v aplikačnom stave. Napríklad uloženie výsledku odpovede zo servera do nášho aplikačného stavu.

Keď spojíme tieto dva koncepty, može sa stať, že uživateľ vloží nejaké dáta a uloží ich, zatiaľ čo na serveri stále pretrváva niečo iné, čo spôsobuje nekonzistentný stav.

Redux sa snaží urobit zmeny stavu predvídateľné, bez stratenia výkonu v podobe asynchronicity. Robí to vynútením určitých podmienok, ako môže zmena stavu nastať.

Tieto podmienky robia aplikáciu predvídateľnú a ľahko testovateľnú. Uľahčuje vývoj možnosťou prechádzať medzi jednotlivými stavmi aplikácie počas ladenia. Túto funcionalitu je možné použiť aj v produkčnom prostredí, kedy uživatelia reportujú chybu a popri tom môže byť prenesený celý aplikačný stav. To znamena, že vývojár môže kedykoľvek načítať totožný stav aplikácie, v ktorom nastala chyba a jednoducho ju reprodukovať.

3.2 Rozhranie pre prístup k médiám

Jedna z technológií, ktorá v posledných rokoch prišla do webových prehliadačov je schopnosť pristupovať ku kamere a mikrofónu bez potreby doplnkov tretích strán. Vďaka `Streams API` a jej metóde `getUserMedia()` je to možné bez doplnkov. Táto metóda vyzve používateľa na povolenie používať vstup médií, ktorý vytvára `MediaStream` obsahujúci vyžiadané typy médií. Tento stream môže obsahovať video stopu, zvukovú stopu a možnosť iných typov stôp.

3.2.1 Podpora webových prehliadačov

Podpora pre `getUserMedia` metódu je v prehliadačoch už nejakú dobu. Prvotná podpora sa začala objavovať v roku 2012, kedy mali prehliadače vytvorené svoje vlastné rozhranie, špecifické pre daný prehliadač. Od roku 2016 sa začali objavovať nové verzie prehliadačov s podporou štandardného rozhrania. Podľa štatistík [4] podporuje túto metódu priamo 91.52 % mobilných prehliadačov a 1.43 % prehliadačov podporu má, ale neštandardným prístupom. Prehliadače pre osobné počítače sú na tom veľmi podobne. Konkrétne 89.08 % s priamou podporou a 2.94 % neštandardným prístupom. Jednotky percent nepodporovaných prehliadačov sú spôsobené tým, že niektorí používatelia stále používajú prehliadače v niekoľko rokov starej verzii. Väčšina prehliadačov pre osobné počítače má priamu podporu od roku 2016, pri prehliadačoch pre mobilné zariadenia je to od roku 2018.

Kapitola 4

Návrh riešenia

V nasledujúcich podkapitolách je opísaný návrh aplikácie postupne od databázového cez návrh REST aplikačného rozhrania, až po samotné uživateľské rozhranie vzhľadom na požiadavky zadania.

4.1 Databázová vrstva

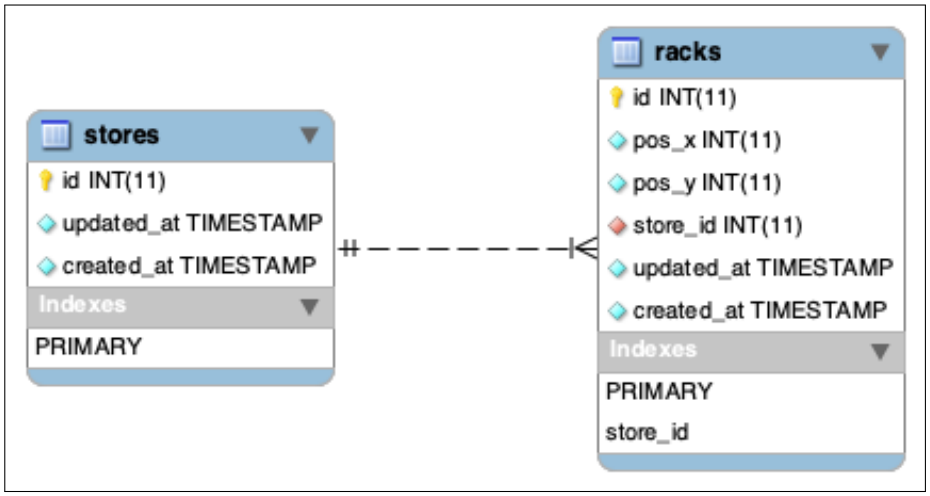
Použitý bol databázový systém MySQL, podľa zadania práce. Každý sklad sa skladá z mapy, čo je v našom prípade obrázok. Obrázok môžeme chápať ako atribút skladu. Každý sklad je zložený z regálov, ktoré majú v rámci skladu určenú polohu. Na určenie polohy nám stačia 2 súradnice (X, Y). Relácia medzi skladom a regálmi je znázornená na obrázku 4.1. Obidve súradnice sú z databázového hľadiska celé čísla, ktoré je nutné mať definované, pretože nie je možné vytvoriť regál bez polohy.

Každý regál obsahuje určený počet políc. Avšak, počet políc nie je iba atribút regálu, ale polica je samostatná entita s vlastnými atribútmi. Jej atribútmi sú umiestnenie v rámci regálu a počet riadkov a stĺpcov, do ktorých je možné umiestniť produkt. Relácia medzi regálom a policami je znáznornená na obrázku 4.2.

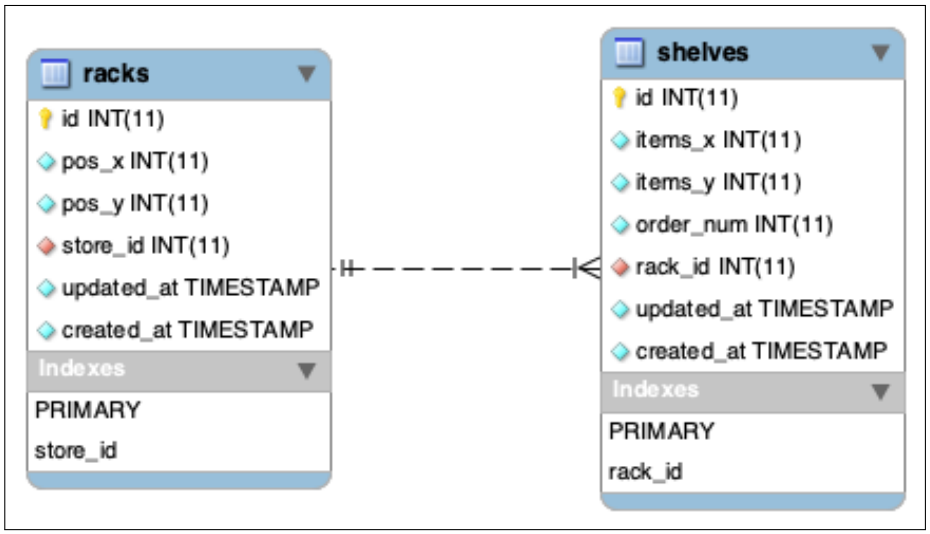
Pre sklad sú definované kategórie produktov, ktoré v ňom budeme skladovať. Kategória má svoj názov a farbu, na základe ktorej môžeme produkty jednoduchšie rozlíšit pri pohľade na policu.

Na policiach sú umiestňované produkty, ktoré majú svoju pozíciu v rámci police, ale produkt môže byť vytvorený aj bez nej. Pozíciu produktu je možné definovať až dodatočne. K produktu musí byť definovaná kategória. Kategórie majú k sebe definované atribúty, kde každý z nich má názov, typ hodnoty a či sa daný atribút bude vyskytovať na vygenerovanom QR kóde. Produkt priradený ku kategórií má nastavené jej atribúty. Tu nastáva problém, že počet potenciálne použitých atribútov pre produkt je veľký, ale reálne použitých atribútov na jednotlivý produkt bude relatívne malý. V takomto prípade máme niekoľko možných riešení:

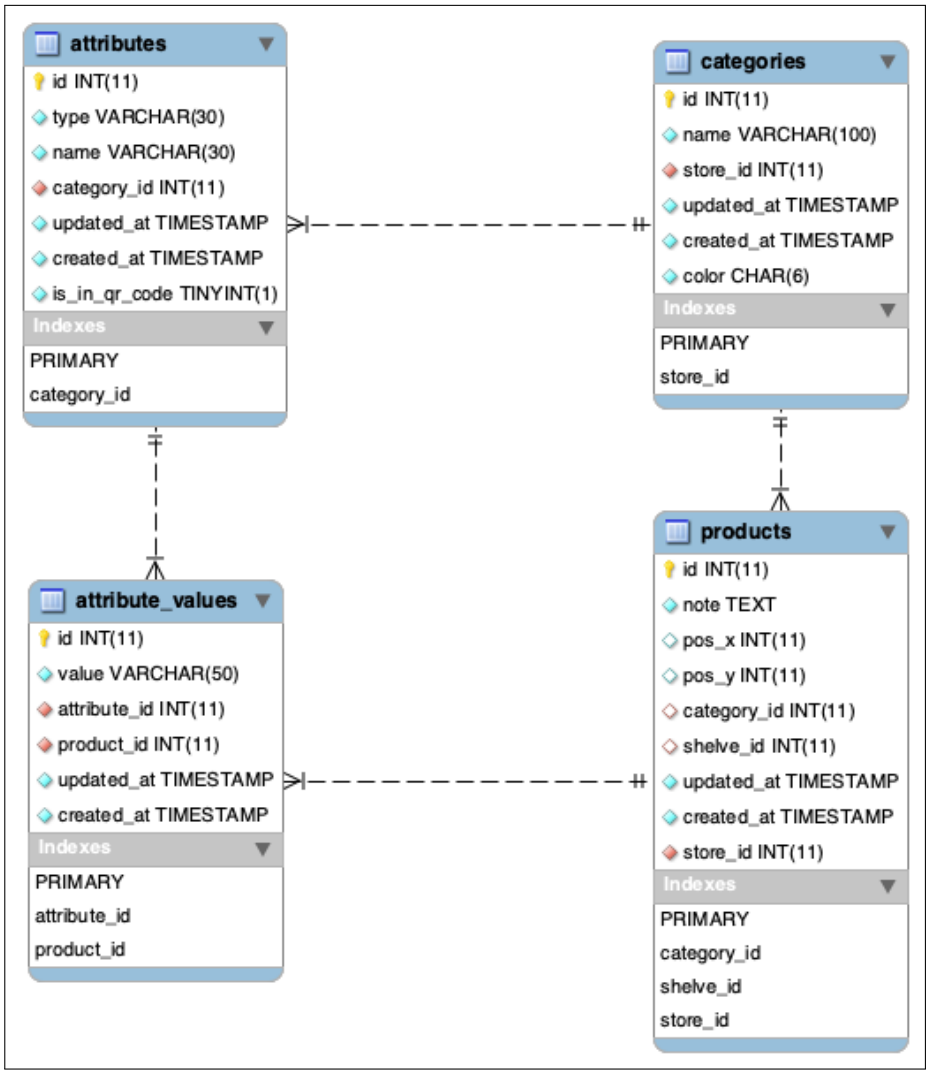
1. EAV (entity attribute value) model, pri ktorom je reprezentácia dát analogická ku vzhľadom na miesto efektívnym metódam na ukladanie riedkych matíc, kde sú uložené iba neprázdne hodnoty. Každý pár atribút-hodnota opisuje entitu a každý záznam v EAV tabuľke má v sebe uloženú práve jednu hodnotu. Data sú uložené v troch stĺpcoch - entita, atribút a hodnota.



Obr. 4.1: Relácia medzi skladom a regálmi.



Obr. 4.2: Relácia medzi regálom a policami.



Obr. 4.3: Relácia medzi kategóriou, produktom a atribútmi.

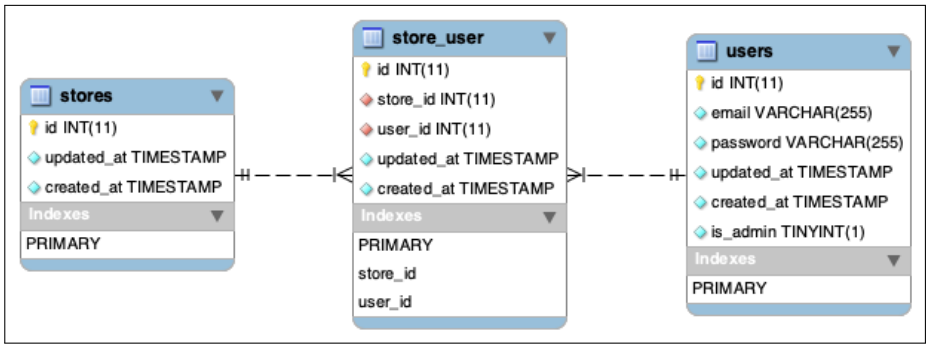
2. Relačný model je viac priamočiarý spôsob ako EAV. V tomto prípade znamená pridanie nových atribútov zásah do schémy. Každý atribút by bol samostatný stĺpec, čo by znamenalo, že pri väčšom množstve atribútov, by bola väčšina pre jednotlivé záznamy nevyužitá.

Druhé riešenie je v našom prípade nevyhovujúce, pretože by aplikácia stratila flexibilitu a nové atribúty by bolo možné pridávať iba zásahom databázového administrátra.

Relácia medzi kategóriou, jej atribútmi, produktom a nastavenými atribútmi produktu je znázornená na obrázku 4.3.

4.1.1 Uživateľské prístupy

Na databázovej vrstve je vytvorená entita `users` s atribútmi email, hash hesla a označením či je daný používateľ administrátor. Email je unikátny, kvôli zabráneniu duplicity používateľov. Používateľ je v relácií so skladom N:M, kde každý používateľ musí byť v relácií so



Obr. 4.4: Relácia medzi používateľmi a skladmi.

skladom, aby k nemu mal prístup (v aplikačnom rozhraní má každý používateľ s označením administrátora prístup ku všetkým skladom). Reláciu je možné vidieť na obrázku 4.4.

4.2 Aplikačná vrstva

Použitý jazyk pre aplikačnú vrstvu je PHP 7, ako hovorí zadanie práce. Návrh aplikačnej vrstvy pozostáva najmä z navrhnutia API endpointov, pri ktorého tvorbe sa budeme držať určitých zasád opísaných v nasledujúcom texte.

4.2.1 Návrh API endpointov

Je niekoľko konvencií, ktoré budeme dodržiavať pri návrhu nášho REST API. API by malo byť navrhnuté vzhľadom na zdroje, čo môžu byť entity alebo služby, takže to vždy musia byť podstatné mená. Napríklad, namiesto `/vytvorUzivatela` použijeme `/uzivatelia`.

Pre názvy zdrojov budeme používať množné čísla. Neexistuje žiadne pravidlo, kvôli ktorému by sme nemohli používať jednotné číslo. Dôvod použitia množného čísla je, že operujeme s jedným zdrojom z kolekcie zdrojov. Preto kvôli vyjadreniu toho, že pracujeme s kolekciou, použijeme plurál.

Ako už bolo naznačené, URL nebudú obsahovať žiadne slovesá. Definovať akcie, ktoré budú na zdrojoch vykonané, budú HTTP metódy.

Príklad ako by mohla vyzerať práca s kategóriami cez API je ukázané v tabuľke 4.1.

Pre zjednodušenie enkódovania identifikátorov zdrojov v URL, musí ich reprezentácia pozostávať iba z ASCII reťazcov z písmen, čísiel, podtržítka, mínusu, dvojbodky a bodky.

Niektoré API zdroje môžu obsahovať alebo referencovať podzdroje. Sú to zdroje, ktoré nie sú na vrchnom leveli zdrojov, ale tvoria časti niektorého z vyššie postavených zdrojov a nemôžu byť použité mimo jeho pôsobnosti.

Zložené identifikátory nemôžu obsahovat `/` ako separátor. Kvôli zlepšeniu skúsenosti uživateľov, by sme mali používať intuitívne zrozumiteľné URL, kde každá časť cesty je validný odkaz na zdroj, alebo množinu zdrojov. Napríklad, ak `/stores/312/categories/86` je valídna cesta nášho API, potom `/stores/312/categories`, `/stores/312` a `/stores` by mali byť v princípe rovnako validné.

HTTP požiadavky

V nasledujúcom texte si ukážeme, akým spôsobom budeme využívať dotazovacie metódy HTTP.

GET

`GET` požiadavky sú použité na čítanie zdrojov či už jednotlivo alebo množinovo.

- GET požiadavky pre konkrétne zdroje bude generovať 404 v prípade, že zdroj neexistuje
- GET požiadavky pre kolekciu zdrojov môže generovať buď 200 (ak je kolekcia prázdna) alebo 404 (ak kolekcia neexistuje)

PUT

`PUT` požiadavky sú použité na úpravu celých zdrojov.

- `PUT` požiadavky sú zvyčajne aplikované na jeden konrétny zdroj, nie na kolekciu zdrojov, pretože to by znamenalo náhradu celej kolekcie
- Po úspešnej `PUT` požiadavke server nahradí celý zdroj adresovaný podľa URL, reprezentáciou uvedenou v tele požiadavky
- Úspešná `PUT` požiadavka bude zvyčajne generovať 200 alebo 204 (ak bol zdroj zmenený - s alebo bez samotného zdroja v odpovedi) a 201 (ak bol zdroj vytvorený)

Nie je vhodné používať `PUT` metódu na tvorbu zdrojov. Toto necháva identifikátor zdroja pod kontrolou samotnej služby.

POST

`POST` požiadavky používame na tvorbu jednotlivých zdrojov cez URL odkazujúcu na kolekciu zdrojov.

- Po úspešnej `POST` požiadavke, server vytvorí jeden alebo viacero zdrojov a vráti ich identifikátory v odpovedi.
- Úspešné `POST` požiadavky zvyčajne generujú 200 (ak bol zdroj upravený), 201 (ak bol zdroj vytvorený), 202 (ak bola požiadavka akceptovaná, ale zatiaľ nie je dokončená) a výnimočne 204 s `Location` hlavičkou (v prípade, že zdroj nie je súčasťou odpovede)

PATCH

`PATCH` požiadavky používame na zmenu častí jednotlivých zdrojov, kde je nahradená iba špecifická podmnožina atribútov zdroja. Sémantika požiadavky nie je definovaná v HTTP štandarde a musí byť opísaná v API špecifikácií pomocou vhodných typov médií.

- `PATCH` požiadavky sú zvyčajne aplikované na jednotivé zdroje, pretože zmena celých kolekcií by bola náročná
- `PATCH` požiadavky zvyčajne nie sú odolné voči chýbajúcej inštancii zdroja
- Úspešné `PATCH` požiadavky upravia časti zdroja adresovaného podľa URL a definovanej zmeny, ktorá je časťou tela požiadavky

- Úspešné `PATCH` požiadavky zvyčajne generujú 200 alebo 204 (if bol zdroj upravený s alebo bez upraveného obsahu v odpovedi)

Keďže použitie `PATCH` je trochu zložitejšie, je lepšie zvoliť jeden z nasledujúcich vzorov:

1. Použitie `PUT` s celým objektom na zmenu zdroja, pokiaľ je to možné.
2. Použitie `PATCH` s čiastočným objektom na zmenu častí zdroja. (Toto je JSON Merge Patch, špecializovaný typ média `application/merge-patch+json`, čo je čiastočná reprezentácia zdroja.)
3. Použitie `PATCH` spolu s JSON Patch [2], špecializovaným média typom `application/json-patch+json` obsahujúcim inštrukcie ako zmeniť zdroj.
4. Použitie `POST` (s vhodným popisom požadovanej akcie) namiesto `PATCH`, ak požiadavka nemodifikuje zdroj spôsobom definovaný sémantikou typu média.

V praxi sa JSON Merge Patch [6] ukázal príliš obmedzený. Jeho jednoduchost prichádza s rôznymi limitáciami. Odstránenie vzniká nastavením kľúča na `null`, čo v praxi znamená, že nie je možné nastaviť hodnotu kľúča na `null`. Nemožnosť manipulácie s poliami. Ak chceme pridať element do pola, alebo zmeniť jeden z jeho elementov, je potreba zahrnúť kompletné pole v Merge Patch dokumente, aj keď sú zmeny minimálne.

DELETE

`DELETE` požiadavky používame na mazanie zdrojov.

- `DELETE` požiadavky zvyčajne aplikujeme na jednotlivé zdroje, nie na kolekciu zdrojov, to by znamenalo, zmazanie celej kolekcie
- Úspešné `DELETE` požiadavky zvyčajne generujú 200 (ak je zmazaný zdroj súčasť odpovede) alebo 204 (ak nie je súčasťou odpovede)
- Neúspešné `DELETE` požiadavky zvyčajne generujú 404 (v prípade, že zdroj nebol nájdený) alebo 410 (ak bol zdroj už zmazaný)

Po zmazaní zdroja pomocou `DELETE`, `GET` požiadavky pre rovnaký zdroj vráti v odpoved 404 (nenájdený) alebo 410 v závislosti od toho, ako je zdroj reprezentovaný po zmazaný. Za žiadnych podmienok, zdroj nemôže byť po tejto operácií dostupný.

Implicitné filtrovanie zdrojov

Niekedy určité kolekcie zdrojov alebo dotazy nebudú ponúkať zoznam všetkých možných elementov, ktoré obsahujú, ale iba tie, pre ktoré má súčasný klient autorizovaný prístup.

Napríklad, ak má jeden užívateľ prístup k trom skladom a vykoná `GET /stores`, nemôžu sa mu zobraziť sklady, ku ktorým nemá prístup alebo nie su pod jeho správou. Tento užívateľ by nikdy nemal vidieť, že spravujeme aj sklady iným užívateľom (nemal by vidieť sklady, ktoré sám nespravuje).

Tabuľka 4.1: Použitie HTTP metód.

Zdroj	GET	POST	PUT	DELETE
/categories	Odpovie listom kategórií	Vytvorí novú kategóriu	Upraví množinu kategórií	Zmaže všetky kategórie
/categories/123	Odpovie špecifickou kategóriou	Nepovolená metóda (405)	Upraví špecifickú kategóriu	Zmaže špecifickú kategóriu

Povolenie voliteľné vloženie podzdrojov

Vkladanie súvisiacich zdrojov (alebo tiež expanzia zdrojov) je dobrý spôsob, ako minimalizovať počet požiadaviek. V prípadoch, keď klient dopredu vie, že bude potrebovat nejaký súvisiaci zdroj, môže dať pokyn serveru, aby tieto zdroje načítal spolu z hlavným zdrojom. Či už je to optimalizované na serveri napr. databázovým `JOIN`, alebo generickým spôsobom, napr. HTTP proxy, ktorá vkladá zdroje, je už vec implementácie.

4.2.2 Autentifikácia a autorizácia endpointov

API chceme mať zabezpečené, aby nemal každý, kto má dokumentáciu prístup ku všetkým dátam. Pri **autentifikácií** sa určuje identita klienta a pri **autorizácií** sa overuje či má klient oprávnenie na vykonávanie požadovanej akcie.

Jednoduché overenie prístupu

Najjednoduchší spôsob ako zabezpečiť API endpointy je HTTP Basic Authentication. V tomto prístupe klient odošle meno a heslo na preukázanie overenia. Toto riešeni nepožaduje cookies, relačné ID, alebo iné špeciálne riešenia, pretože používa HTTP hlavičku. Problém je, že autentifikácia je prenášaná nezabezpečene a je to možné zneužiť napríklad pomocou `man in the middle` útoku.

JSON Web Token

V tomto spôsobe je vyriešený problém HTTP Basic Authentification a jeho posielanie nezabezpečeného hesla na server. Klient sa prihlási cez autentifikačný server použitím mena a hesla (prípadne Facebook login, Google login, ..). Autentifikačný server vytvorí JWT (JSON Web Token) [8] a odošle ho v odpovedi klientovi. Keď klient robí ďalšie požiadavky na server, odosiela spolu s nimi aj JWT. Server v tomto prípade musí vedieť verifikovať, že prichádzajúci JWT je vytvorený autentifikačným serverom.

4.3 Prezentačná vrstva

Klientská časť je vytvorená pomocou javascriptovej knižnice React. V porovnaní s možnosťou vytvárania natívnej mobilnej aplikácie, prináša toto riešenie výhodu v podobe prenositeľnosti aplikácie. Aplikácia je webová, čo znamená, že je možné ju využívať osobnom

Obr. 4.5: Horná lišta.

počítači a aj na mobilných zariadeniach s prehliadačom podporujúcim Javascript. Z frontendových frameworkov je použitý React, pretože v porovnaní s Vue má oveľa väčšiu komunitu (s tým spojenú podporu a veľa zdrojov) a v porovnaní s Angularom, nie je tak viazaný na písanie špecifického kódu pre framework[5].

4.3.1 Prihlasovací formulár

Prihlasovací formulár funguje, ako vo väčšine webových aplikáciách. Jednoduchý formulár s dvomi položkami pre zadanie emailu a hesla, pod ktorým sa nachádza odkaz na registračný formulár, určený pre nových používateľov.

4.3.2 Hlavná lišta

Dôležité je si uvedomiť, ktoré akcie bude používateľ vykonávať najčastejšie a sprístupniť ich tak, aby bolo preňho jednoduché sa k nim dostať. Súčasťou hlavnej lišty je tlačidlo pre otvorenie hlavného menu (obrázok 4.5). Podľa práve vybranej položky sa mení nadpis umiestnený na lište, takže uživateľ vždy vie, kde sa nachádza. Tlačidlá uložené na pravej strane lišty sa menia podľa kontextu aplikácie. Ak sa nachádzame v správe skladu, tlačidlá nám ponúkajú možnosť vytvoriť nový sklad, alebo vybrať iný. V prípade položky kategórie sa nám zobrazí tlačidlo, pomocou ktorého sa dostaneme do formulára vytvorenia novej kategórie. Pri produktoch môžeme vidieť tlačidlá pre vytvorenie nového produktu, skenovanie QR kódu a zobrazenie filtrov pre práve zobrazený list produktov. V hlavnom menu sa nachádza aj tlačidlo pre odhlásenie a v prípade administrátorského účtu aj položka na správu účtov.

4.3.3 Mapa skladu

Pre vykreslenie mapy skladu máme niekoľko požiadaviek.

- Musíme byť na ňu schopný dodatočne vykresliť regále
- Mať schopnosť približovania a vzďalovania sa
- Presúvať sa cez mapu pomocou dotyku (dá sa predpokladať, že vo väčšine prípadov sa mapa celého skladu nezmestí na obrazovku zariadenia tak, aby bolo možné ju bez problémov ovládať)

HTML5 poskytuje element Canvas, ktorý ponúka dynamické, skriptovateľné rendrovanie 2D tvarov a bitmapových obrázkov. Je to nízkoúrovňový, procedurálny model upravujúci bitmapu, bez zabudovaného grafu scény (pomocou WebGL podporuje 3D tvary, obrázky a podobne). Pozostáva z oblasti na kreslenie definovanej HTML kódom s atribútmi výška a šírka. Javascriptový kód môže pristúpiť k tejto oblasti cez množinu kresliacich funkcií, podobných iným 2D aplikačným rozhraniam a umožňuje nám dynamicky generovanú grafiku. Canvas sa bežne používa na vytváranie grafov, animácií, hier a kompozícií.

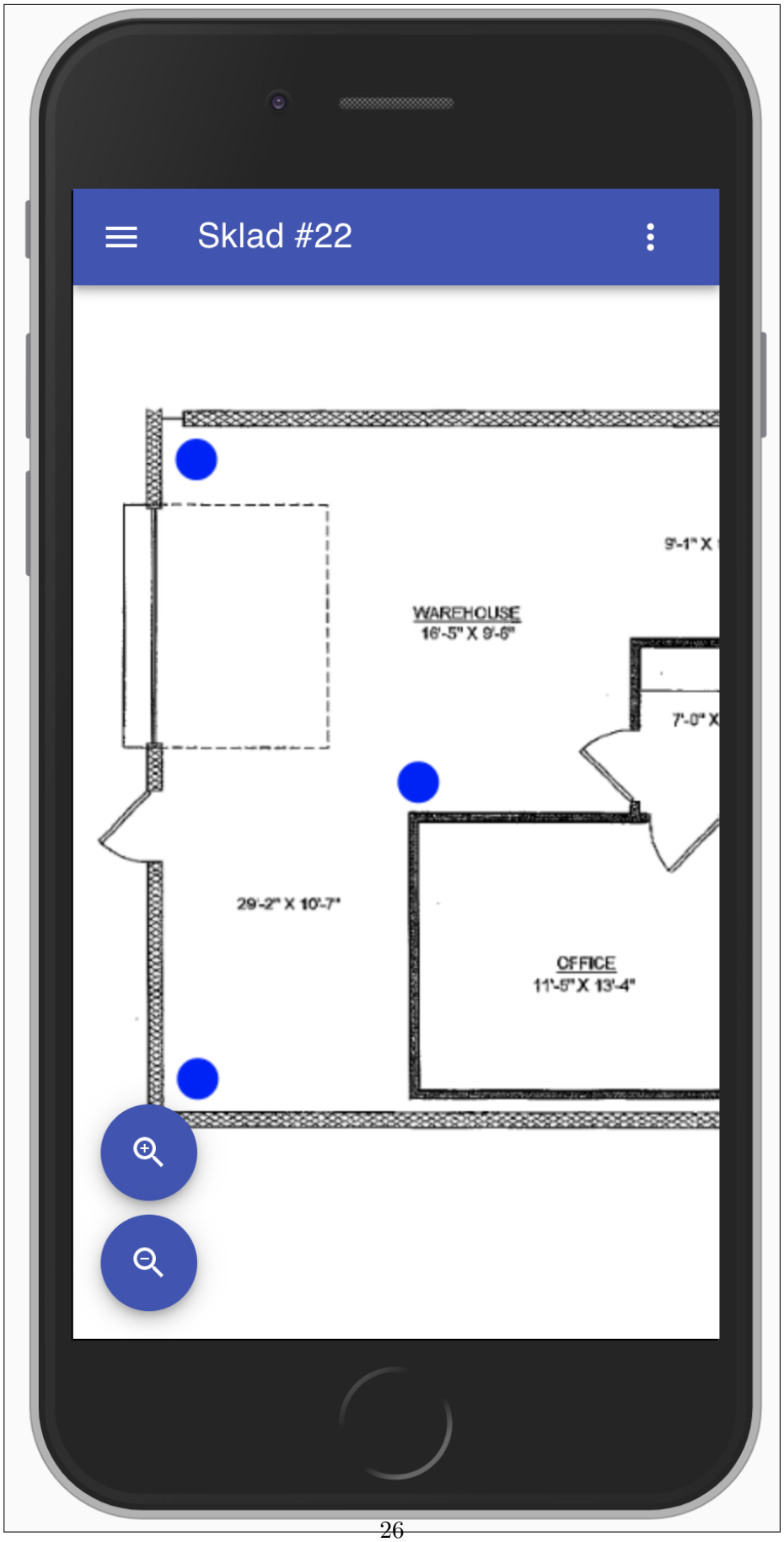
V našom prípade na Canvas v prvom rade vykreslíme mapu skladu vo forme obrázku a následne naňho budeme kresliť regále. Regále sú vykreslené pomocou jednoduchého tvaru (kruhu), znázornené na obrázku 4.6. Používateľ sa môže pohybovať po mape dotykmi prstu, alebo operáciou potiahni a pusť ovládanou myšou. Približovanie bude umožnené pomocou tlačidiel v rohu obrazovky, kde budú umiestnené zvlášť tlačidlo na priblíženie a oddialenie zobrazenia. Mapa bude reagovať na dotyk a na kliknutie, aby bolo zabezpečené ovládanie jak z mobilného zariadenia, tak z osobného počítača. Po kliknutí, alebo po dotyku na existujúci regál sa vykreslí detail regálu. Ak sa na vybranom mieste regál nenachádza, zobrazí sa okno určené k vytvoreniu nového regálu.

4.3.4 Vykreslenie regálu

Pri pohľade na regál by sme v ňom mali byť schopný nájsť požadovaný produkt, alebo minimálne zistiť, aké kategórie produktov v ňom máme. Ak nás niektorý z produktov zaujíma, musíme mať možnosť si tento produkt prehliadnuť rovnako, ako pri skutočnom regáli. Problém znova nastáva, keď je produktov veľa a nie je možné ich zobraziť naraz. Rovnako je potrebné znázorniť, na ktorej poličke sa produkt nachádza. Preto bude lepšie zobrazenie zjednodušiť a obmedziť ho iba nazobrazenie jednej poličky s možnosťou presúvania sa medzi nimi. Presúvanie medzi poličkami je možné urobiť štyrmi spôsobmi:

1. Posunom prstu smerom hore, alebo dolu, kde každý posun znamená, posunutie na ďalšiu poličku daným smerom.
2. Posunom prstu smerom hore, alebo dolu, kde veľkosť pohybu určuje, o koľko políc sa posun vykoná.
3. Dotykom na poličku, ktorú chceme otvoriť.
4. Dotykom na vrchnú a spodnú časť obrazovky, kde by každý z týchto dotykov znamenal posun o jednu poličku hore, respektíve dolu.

Prvý spôsob umožňuje pohodlné posúvanie prstom, kde skladník môže intuitívne aj bez pozerania sa potiahnuť prstom po obrazovke a posunúť sa o poličku vyšie respektíve nižšie. Pri druhom spôsobe by sa používateľ musel naučiť citlivosť posunutia a musel by vždy sledovať, pri veľkom počte políc možno dokonca odrátavať či sa naozaj posunul na tú správnu policu. Dotyk na poličku, ktorú chceme otvoriť by nám umožňoval rýchly presun cez poličky a priame otvorenie poličky, ktorú chceme práve prezerať. Znamenalo by to však, že každá polička by musela byt vykreslená dostatočne veľká, aby bolo možné na nu kliknúť, bez pomýlenia sa. Pri zmenšenom zobrazení je možné riziko, že užívateľ klikne napríklad na produkt. Štvrtý spôsob ponúka jednoduchý spôsob posúvania, presným počtom kliknutí by sa užívateľ dokázal dostať presne tam, kam potrebuje, ale znova hrozí, že na mieste kliku bude umiestnený produkt. Preto som zvolil prvý spôsob, pri ktorom nehrozí, že by sme spustili nejakú inú, nežiadanú akciu.



Obr. 4.6: Nákres skladu.

Na každej poličke sú vyznačené miesta, na ktoré je možné umiestniť produkt (prípadne už na tomto mieste môže byť produkt umiestnený). Počet riadkov a stĺpcov je možné meniť. Predvolený počet stĺpcov je 5 a riadkov 3. Aj keď na poličke ešte žiadne produkty nie sú, keby je počet stĺpcov aj riadkov nastavený na 1, nedokázali by sme povedať či je produkt v strede, na kraji, alebo niekde medzi. Toto nám umožní predvolený počet stĺpcov 5. V prípade, že nám 5 stĺpcov nestačí, pri pridávaní nových produktov môžme vytvoriť nové.

4.3.5 Správa kategórií

Kliknutím na položku Kategórie z bočného menu sa dostaneme do okna správy kategórií. V správe kategórií máme k dispozícií list už existujúcich kategórií a tlačidlo na vytvorenie novej. Po výbere jednej z existujúcich kategórií sa nám zobrazia jej atribúty (názov, farba) a atribúty, ktoré majú definované produkty danej kategórie. Pri týchto atribútoch je možné zmeniť všetko okrem ich dátového typu. Dátové typy sú 4 - celé číslo, desatinné číslo, text a dátum. Tieto typy by mali pokryť všetku potrebu nášho skladu. Pri vytváraní novej kategórie definujeme jej názov, farbu a počet atribútov upravujeme tlačidlom v rohu. Po pridaní atribútu sa na koniec strany pridá nový atribút, ktorému definujeme meno, typ a či má byť súčasťou vytlačeného QR kódu.

4.3.6 Správa produktov

Pridávanie

Pridanie produktu je možné spustiť z hlavnej lišty. Pri každom novom produkte musíme nastaviť kategóriu a po jej výbere sa nám zobrazia nastaviteľné atribúty prislúchajúce k danej kategórií. Produkt bude možné vytvoriť (odoslať požiadavku na server) iba v prípade, že všetky atrubúty sú valídne vzhľadom na ich typ. Pre každý typ musí byť definovaný validátor na strane klienta (nemusí, ale v opačnom prípade by sa používateľ dozvedel, že je niečo zle, až po odoslaní požiadavky na server). Počas celej doby vytvárania produktu bude možnosť definovať jeho umiestnenie. Avšak, produkt môžme uložiť aj bez umiestnenia. Ak je v sklade nejaký produkt s nedefinovaným umiestnením, aplikácia nás na to upozorní pri zobrazení regálu, kde môžme takýto produkt umiestniť. V rohu sa nám rozsvieti tlačidlo a po kliknutí si môžeme vybrať, ktorý z neumiestnených produktov chceme vybrať.

Spôsob umiestnenia

Umiestnenie produktu možno riešiť rôznymi spôsobmi. Jeden z nich je, že si nájdeme regál, poličku a konrétne miesto na ktoré klikneme a vtedy sa nám spustí tvorba produktu. Toto ale nevyhovuje požiadavke, že produkt môže byť vytvorený aj bez určeného miesta v sklade. Produkt je definovaný v grafickom rozhraní aplikácie ako kruh, ktorý je farebne odlíšený podľa svojej kategórie. Produkt by sme mohli aj dodatočne umiestniť len kliknutím na miesto. Problém je takýmto spôsobom umiestniť produkt na miesto, ktoré nie je definované. Napríklad, keď už máme dva produkty vedľa seba a tretí chceme umiestniť medzi ne. Preto som zvolil spôsob umiestňovania s využitím potiahnu a pusť funkcie. Týmto spôsobom dokážeme dosiahnuť podobných vecí, ako na osobnom počítači sledovaním miesta kurzora a či sa nachádza nad určitým miestom. Neumiestnený produkt chytíme, potiahneme na miesto, kam ho chceme umiestniť a pustíme. Pri prechádzaní cez poličku sa vykresluje, na ktoré miesto by bol produkt umiestnený v prípade pustenia.

Zobrazenie umiestnenia

Ak má produkt definované umiestnenie, pri zobrazeni jeho detailu je možnosť na zobrazenie jeho umiestnenia tlačidlom umiestneným na hlavnej lište. Po kliknutí sa zobrazí mapa skladu so zvýrazneným regálom, v ktorom sa daný produkt nachádza. Kliknutim na regál sa nám zobrazí detail regálu s otvorenou poličkou a zvýrazneným umiestnením produktu. V tomto móde sú všetky ostatné akcie pri zobrazení skladu a regálu vypnuté a slúži iba na zobrazenie umiestnenia.

Verifikácia produktu

Pri zobrazení detailu produktu potrebujeme možnosť verifikovať či sa produkt otvorený v aplikácií a reálny produkt v sklade zhodujú. Preto je v detaile produktu umiestnené tlačidlo na verifikáciu, ktorým spustíme kameru na načítanie QR kódu a uvidíme či sa QR kód zhoduje s práve otvoreným produktom, alebo nie.

4.3.7 Vyhľadávanie

Ďalšia z položiek v hlavnej lište pri zobrazení listu produktov je tlačidlo na zobrazenie filtrov. Podľa zadania potrebujeme vedieť filtrovať a zoradiť produkty podľa ich dátumu spotreby a výroby. Preto sa vo filtračnom okne nachádzajú pre každý z týchto dátumov dve položky. Jedna je definovanie minimálneho a druhá maximálneho dátumu, ktorý nás zaujíma. Zoradiť produkty môžme taktiež podľa týchto dvoch dátumov, s voľbou smeru zoradenia.

4.3.8 Tlač QR kódu

Tlač QR kódu je jedna z položiek hlavného menu, ktorá z veľkej časti ponúka funcionalitu vyhľadávania. Pri tomto je možné označiť produkty, ku ktorým práve chceme vygenerovať QR kód. QR kódy by mali byť vygenerované v mriežke, aby ich následne bolo možné vytlačiť na delený lepiaci papier. Pred tlačením má užívateľ možnosť nadefinovať mriežku rozdelenia papiera formátu A4.

4.3.9 Administrácia

Administrátor má v porovnaní s bežnými účtami 2 rozdieľne veci. Prvou je prístup ku všetkým existujúcim skladom a tou druhou je položka menu Správa účtov. V správe účtov môžeme udeľovať, prípadne odoberať prístupy jednotlivým používateľom ku skladom. Po založení účtu má každý používateľ prístup iba ku svojim vlastným skladom.

Kapitola 5

Implementácia

Táto kapitola je venovaná samotnej implementácií aplikačného rozhrania a s ním spojeného REST API použitím frameworku Laravel a v druhej časti sa bude venovať implementácií klientskej strany – uživateľského rozhrania, implementovanej v jazyku Javascript za použitia knižnice ReactJS a Redux.

5.1 Implementácia REST API

5.1.1 Štruktúra aplikácie

Štruktúra aplikácie je pevne daná šablónou po vytvorení Laravel projektu cez Composer aplikáciu. V zložke `app` sú vytvorené jednotlivé modely korešpondujúce s databázovými tabuľkami. Šablóna modelu je vytvorená pomocou príkazu `php artisan make:model`, ktorý automaticky vytvorí model, ktorý je na základe mena triedy naviazaný na danú tabulku (tabuľky sú v množnom čísle a modely v jednotnom). Každý model bez dodatočnej konfigurácie predpokladá existenciu stĺpcov `created_at` a `updated_at`. Kedže v žiadnom z endpointov nepovolujeme definovanie identifikátorov klientovi, všetky modely majú nastavený atribút `$guarded`. Tento atribút obsahuje list atribútov, ktoré nemôžu byť priradené. Každý model má ďalej definované relačné funkcie, pomocou ktorých vieme cez model pristúpiť k relačným modelom.

V zložke `app/Http/Controllers` sú vytvorené kontroléry pre jednotlivé zdroje. Kontrolér je definovaný, ako trieda rozširujúca (dediaca) z triedy `Controller`. Funkcie kontrolérov predstavujú obsluhu konrétnych endpointov.

Definícia endpointov

Endpointy sú definované v `routes/api.php`. Každý endpoint je definovaný HTTP metódou, URL cestou a konkrétnou metódou kontroléru, ktorá obsluhuje požiadavku. Príklad definície endpointu je `Route::post('/store', 'StoreController@create')`, kde je URL cesta `/store`, obsluhujúca metóda `create` sa nachádza v kontroléri `StoreController` a použitá HTTP metóda je `POST`. Endpointy sú rozčlenené do skupín podľa toho či potrebujú autorizáciu alebo nie. Skupina endpointov je vytvorená príkazom `Route::group` a obsahuje špecifický middleware pre danú skupinu. Skupiny sú 4:

1. Endpointy, pre ktoré nie je potrebná žiadna autentifikácia (napríklad endpoint na registráciu nového účtu)

2. Endpointy, pre ktoré je potrebné byť autentifikovaný, ale nie je potrebná autorizácia (endpointy týkajúce sa vytvárania skladu).
3. Endpointy, pre ktoré musí byť uživateľ autorizovaný (všetky endpointy týkajúce sa správy skladu).
4. Endpointy, pre ktoré je autorizovaný iba admin (správa prístupov ku skladu)

Kontroléry

Každá metóda triedy kontroléru je použitá, ako obslužná metóda pre konkrétnu metódu endpointu. Vytvorené kontroléry sú:

- AuthController.php - obsahuje metódy určené pre prihlasovanie, registráciu, odhlásenie, informácie o práve prihlásenom používateľovi a obnovu autorizačného tokenu.
- CategoryController.php - obsahuje metódu pre načítanie kategórií ku konkrétnemu skladu, načítanie kategórie podľa identifikátor, vytvorenie, úpravu a mazanie.
- ParameterController.php - obsahuje metódu na načítanie parametrov podľa kategórie, vytvorenie, úpravu a zmazanie parametra.
- ProductController.php - obsahuje metódu na načítanie produktov podľa poličky, načítanie neumiestnených produktov, načítanie produktu podľa itentifikátora, vytvorenie, úpravu a mazanie produktu.
- RackController.php - obsahuje metódu na načítanie regálov podľa skladu, vytvorenie, úpravu a zmazanie regálu.
- ShelveController.php - obsahuje metódy na vytvorenie, úpravu, mazanie a načítanie poličiek.
- StoreController.php - obsahuje metódu na vytvorenie a načítanie sklada a na načítanie všetkých skladov, ku ktorým má prihlásený používateľ prístup.

Validácia

Pre validáciu je použité Laravel Form Request. Sú to špecifické triedy požiadaviek, obsahujúce validačnú logiku. Na vytvorenie takejto triedy možno použiť `make:request` Artisan CLI príkaz:

```
php artisan make:request NewCategory
```

Výpis 5.1: CLI príkaz na vytvorenie triedy

Vygenerovaná trieda je umiestnená v zložke `app/Http/Requests`. Do vygenerovanej metódy `rules` pridáme validačné pravidlá:

```
/**
 * Get the validation rules that apply to the request.
 *
 * @return array
 */
public function rules()
```

```
{
    return [

    ];
}
```

Výpis 5.2: Metóda pre validačné pravidlá

Do obslužnej metódy sme pridali požiadavku typu `NewCategory`. Prichádzajúca požiadavka je validovaná predtým, ako je zavolaná obslužná metóda, čo znamená, že v kontroléroch nebude obsiahnutá žiadna validačná logika.

Vygenerovaná trieda obsahuje aj metódu `authorize`, určenú pre kontrolu či má autentifikovaný používateľ práva na vykonanie danej požiadavky. Táto logika je v našej aplikácií vyriešená pomocou middlewaru a preto vždy vraciame `true` z metódy `authorize`.

Autorizácia

Autorizáciu používateľov sme vyriešili pomocou middlewaru, ktorý je zavolaný vždy ešte pred samotnou obslužnou metódou k požiadavke. Middleware môžeme vytvoriť Artisan CLI príkazom `php artisan make:middleware Authorize`, ktorý vytvorí nový súbor v priečinku `app/Http/Middleware`. V tomto middleware súbore definujeme metódu `handle`, ktorá po úspešnej autorizácií zavolá ďalší middleware, alebo v opačnom prípade vyvolá výnimku `UnauthorizedHttpException`. V middlewari sa najprv skontroluje či je prihlásený používateľ admin a v prípade, že nie je, skontroluje sa či má prístup ku požadovanému skladu.

```
/**
    * Handle an incoming request.
    *
    * @param \Illuminate\Http\Request $request
    * @param \Closure $next
    * @return mixed
    */
public function handle($request, Closure $next)
{
    $this->authenticate($request);
    $store_id = $request->store_id;
    $store = Store::with('users')->find($store_id);

    if(auth()->user()['is_admin']) {
        return $next($request);
    }

    foreach($store->users as $user) {
        if($user->id == auth()->user()['id']) {
            return $next($request);
        }
    }

    throw new UnauthorizedHttpException('jwt-auth', 'Unauthorized');
```

```
}
```

Výpis 5.3: Middleware zabezpečujúci autorizovaný prístup

Pre endpointy, ktoré môžu používať iba administrátorské účty, sme vytvorili samostatný middleware `AuthorizeAdmin.php`, v ktorý je ochudobnená verzia predošlého middlewaru obsahujúca iba kontrolu pre administrátora.

Využitie databázových transakcií

Každá obslužná metóda kontrolerov pracujúca s databázou, začína vytvorením transakcie a končí zavolaním databázového príkazu `commit`. Vyzerá to následovne:

```
public function create(Request $request, $store_id)
{
    DB::beginTransaction();
    // obsluzny kod
    DB::commit();
}
```

Výpis 5.4: Obslužný kód požiadavky

Databázové transakcie nás v týchto prípadoch chránia pred čiastočne vybavenou požiadavkou, ktorá skončí chybou. Napríklad sa pri vytváraní produktu môže stať, že vytvoríme produkt a pri vytváraní jeho parametrov zistíme, že parameter poslaný v požiadavke neexistuje. Potom keby sme nepoužili transakciu, mali by sme uložený nový produkt, ale požiadavka by skončila chybou, takže žiadny produkt by v tomto prípade vytvorený byť nemal. Buď sa vytvorí produkt aj so všetkými atribútmi jemu prislúchajúcimi, alebo sa nevytvorí nič.

5.1.2 Dokumentácia API

Dokumentácia k REST API je vytvorená v OpenAPI Specification (OAS), pretože definuje štandardné rozhranie pre REST API, neviazané na žiadny programovací jazyk. Umožňuje porozumieť možnostiam služby bez potreby prístupu k zdrojovému kódu, dodatočnej dokumentácie, alebo inšpekcie sieťovej prevádzky.

5.2 Uživateľské rozhranie

Pre vytvorenie React projektu bol použitý nástroj pre spúšťanie `Node` balíčkov - NPX:

```
npx create-react-app store_fe
```

Výpis 5.5: Vytvorenie aplikácie

V priečinku `src` je celý zdrojový kód aplikácie.

5.2.1 Redux

Pridanie reduxu do react aplikácie je možné nainštalovaním balíčkov `redux` a `react-redux` CLI príkazom:

```
npm install redux react-redux
```

Výpis 5.6: Inštalácia balíčkov

Príkaz automaticky pridá ak závislosti do projektu, ktoré sú definované v súbore `package.json`.

Pre lepšiu organizáciu kódu sme vytvorili tri priečinky, `./actions`, `./reducers` a `./store`, všetky v zložke `src`. Pridali sme súbor `./store/configureStore.js` a `./reducers/rootReducer.js`. `rootReducer.js` v sebe kombinuje všetky reduktory pomocou príkazu `combineReducers`, ako môžme vidieť na nasledujúcej ukážke kódu:

```
const rootReducer = combineReducers({
  'access': access,
  'auth': auth,
  'authForm': authForm,
  'categories': categories,
  'category': category,
  'newProduct': newProduct,
  'product': product,
  'products': products,
  'rack': rack,
  'register': register,
  'shelves': shelves,
  'stores': stores,
  'store': store,
  'unassignedProduct': unassignedProduct,
  'unassignedProducts': unassignedProducts,
  'users': users,
});
```

Výpis 5.7: Hlavný reduktor

V `configureStore.js` je definovaná funkcia, ktorá vytvára takzvaný store objekt, ktorý v sebe spája akcie a reduktory. Pri vytváraní tohto objektu mimo iné určujeme, aké middlewary budú aplikované pri vykonávaní každej akcie. Preto, ako jeden z middlewarov použijeme `logger` z balíčka `redux-logger`, ktorý automaticky loguje každú zmenu stavu aplikácie. Ďalším použitým middlewarom je `promise` z balíčka `redux-promise-middleware`, ktorý zjednodušuje a hlavne zjednocuje vytváranie požiadaviek a ich následné spracovanie. Každé vytvorenie akcie sa skladá z typu akcie a prislúchajúcich dát. Napríklad v našej akcii typu `GET_CATEGORIES`, ktorú vytvára nasledujúci kód:

```
export function getCategories(storeId)
{
    return dispatch =>
    {
        dispatch(
        {
            type: "GET_CATEGORIES",
            payload: axios.get(
                'http://127.0.0.1:8000/api/store/' +
                storeId + '/category'),
        })
    }
```

```
}
```

Výpis 5.8: Príklad funkcie slúžiacej na vytvorenie akcie

Tento middleware automaticky vytvori akciu typu `GET_CATEGORIES_PENDING`, čo znamená, že sa požiadavka spracovává a po nej následuje jedna z akcií `GET_CATEGORIES_FULFILLED`, `GET_CATEGORIES_REJECTED`, kde prvá znamená, že požiadavka bola úspešne spracovaná a druhá, že nastala chyba. V našom reduktore na každú z týchto akcií zareagujeme rôznou zmenou stavu.

Akcie

Všetka komunikácia s REST API je definovaná na jednom mieste a to v zložke `actions`. Väčšina akcií slúži na vytvorenie požiadavky na server, prípadne na zmenu stavu uživateľského rozhrania. Nachádza sa tu súbor `actionTypes.js`, ktorý ma v sebe definované všetky typy akcií v podobe konštánt. Ostatné zdrojové súbory obsahujú:

- authActions.js - akcie spojené s registráciou, prihlásením, odhlásením a k nim príslušné akcie pre zmenu stavu uživateľského rozhrania.
- categoryActions.js - akcie pre načítanie kategórií, načítanie konkrétnej kategórie, úpravu a vytvorenie kategórie.
- productsActions.js - akcie pre načítanie produktu podľa poličky, podľa identifikátora, samotné načítanie zatiaľ neumiestnených produktov podľa skladu, vytvorenie produktu, priradenie pozície už existujúcemu produktu, vytvorenie produktu a na zmenu stavu uživateľského rozhrania hovoriacom o práve umiestňovanom produkte.
- rackActions.js - akcie pre vytvorenie regálu a ovládanie stavu uživateľského rozhrania o práve vybranom regáli.
- storeActions.js - akcie pre načítanie všetkých a načítanie jednotlivého skladu.
- userActions.js - akcie pre načítanie používateľov, ich aktuálnych prístupoch a editáciu prístupov.

Reduktory

Reduktory používame na modifikovanie stavu aplikácie. V hlavnom reduktore máme napríklad reduktor `categories`, ktorý je implementačne jednoduchá funkcie, do ktorej vstupuje súčasný stav a aktuálna akcia:

```
const categoriesInitialState = {
  categoriesFetching: false,
  categoriesFetched: false,
  categoriesError: null,
  categories: [],
};

export function categories(state = categoriesInitialState, action) {
  switch (action.type) {
    case GET_CATEGORIES_PENDING:
```

```
      return {
        ...state,
        categoriesFetching: true,
      }
    case GET_CATEGORIES_REJECTED:
      return {
        ...state,
        categoriesFetching: false,
        categoriesError: action.payload,
      }
    case GET_CATEGORIES_FULFILLED:
      return {
        ...state,
        categoriesFetching: false,
        categoriesFetched: true,
        categories: action.payload.data.categories,
      }
    default:
      return {...state};
  }
}
```

Výpis 5.9: Reduktor kategórií

`categoriesInitialState` je prvý stav na začiatku behu aplikácie, ktorý sa do reduktora dostane. Podľa typu akcie tento stav meníme. Dôležité je, že nikdy v reduktore nemeníme objekt súčasného stavu, ale vždy na jeho základe vytvárame nový objekt.

Iným použitím reduktora je napríklad `authForm`, ktorý je súčasťou súboru `authReducer.js`. Je to typický reduktor pre správu stavu používateľského rozhrania, ktorým nastavujeme či je používateľ prihlásený a či sa má zobrazit formulár na prihlásenie alebo formulár na registráciu.

```
const authFormInitialState = {
  showRegisterForm: false,
  showLoginForm: false,
  loggedIn: false,
};

export function authForm(state = authFormInitialState, action) {
  switch (action.type) {
    case SHOW_REGISTER:
      return {
        ...state,
        showRegisterForm: true,
        showLoginForm: false,
      }
    case SHOW_LOGIN:
      return {
        ...state,
        showRegisterForm: false,
```

```
        showLoginForm: true,
      }
    case LOGGED_IN:
      return {
        ...state,
        loggedIn: true,
      }
    case LOGOUT:
      return {
        showRegisterForm: false,
        showLoginForm: true,
        loggedIn: false,
      }
    default:
      return {...state};
  }
}
```

Výpis 5.10: Reduktor autorizácie

Komponenty

Všetky komponenty sú umiestnené v priečinku `src/components`.

Komponent AccessManagement

Slúži na zobrazenie a správu používateľských prístupov. Pracuje s globálnym stavom `access` a `users` a využíva `userActions`. Komponent po nasadení (funkcia `componentDidMount`) spustí načítanie používateľov. Po načítaní používateľov sa komponent znova vyrendruje, čo nám zabezpečí, že budeme mať na výber všetkých aktuálnych používateľov. Po výbere jedného z nich sa nastaví atribút stavu komponentu, ktorý označuje identifikátor používateľa a spustí sa načítavanie práv vybraného používateľa pomocou funkcie z `userActions`. Funkcia `componentDidUpdate` kontroluje či ja načítavanie dokončené následujúcim spôsobom:

```
componentDidUpdate(prevProps)
{
    if (this.props.access.fetched && this.props.access
        .fetching !== prevProps.access.fetching)
    {
        this.setState(
        {
            access: this.props.access.access,
        })
    }
}
```

Výpis 5.11: Ukážka spracovania požiadavky

Po zistení úspešného načítavania sa nastaví stav prístupu komponentu, podľa ktorého sa následne vyrendruje aj jeho zobrazenie.

Komponent AuthContainer
Komponent pracuje s globálnymi stavom `auth`, `authForm` a `register`, pričom používa `authActions`. Po nasadení komponentu sa zistí či je nastavený autorizačný token a ak áno, zistíme ci je stále platný. V opačnom prípade nastavíme zobrazenie prihlasovacej stránky pomocou akcie.

```
componentDidMount() {
  if(this.props.auth.token)
    this.props.authActions.verify();
  else
    this.props.authActions.showLogin();
}
```

Výpis 5.12: Zistenie stavu autorizácie

Pri zobrazovaní podľa globálneho stavu `authForm` zobrazíme buď komponent `RegisterForm`, alebo `LoginForm`. Komponent reaguje na zmenu globálneho stavu, ktorý hovorí o prihlásení, registrácií a verifikácií prihláseného používateľa a podľa toho zobrazuje daný komponent.

Komponent Canvas
Tento komponent pracuje s globálnym stavom `store`, `rack` a `newProduct`, používa pri tom akcie `rackActions` a `categoryActions`. Po nasadení komponentu sa vo funkcii `componentDidMount` nastavujú obslužné funkcie pre uživateľské udalosti, ako je: kliknutie, dotyk, posunutie myšou, posunutie pri dotyku, koniec dotyku a kliku. Pri niektorých z týchto udalostí používame rovnaké obslužné funkcie, ale predtým musíme správne určiť na akej pozícií sa akcia odohrala. Napríklad v prípade udalostí `mousedown` a `touchstart` používame tú istú obslužnú funkciu nasledujúcim spôsobom:

```
startDragOffset.x = evt.type == 'touchstart' ? evt.touches[0]
    .clientX : evt.clientX;
startDragOffset.y = evt.type == 'touchstart' ? evt.touches[0]
    .clientY : evt.clientY;
```

Výpis 5.13: Zistenie stavu autorizácie

Podľa typu udalosti zvolíme spôsob, akým získame pozíciu vykonania udalosti.

Posun zobrazenia – v tejto časti budeme hovoriť iba o dotyku, ale pre prácu s myšou to platí analogicky. Po zachytení udalosti `touchstart` si zaznamenáme pozíciu jej vykonania. Ak nastane udalosť `touchend` a jej pozícia je rovnaká, ako bola pozícia `touchstart` tak vieme, že sa nejedná o pohyb, ale iba o kliknutie. Ak zachytíme udalosť `touchmove`, zmeníme posunutie relatívne podľa pozície zaznamenanej z udalosti `touchstart` a zavoláme funkciu `redraw`, ktorá nanovo vykreslí pôdorys skladu s už aplikovaným posunutím.

Komponent CategoryContainer
Komponent využíva globálny stav `categories`, `category` a `store`, s použitím akcií `categoryActions`. Nasadením komponentu sa spustí načítanie kategórií pre aktuálny sklad.

V prípade tvorby novej kategórie a odoslania požiadavky na server, komponent čaká vo funkcií `componentDidUpdate` na potvrdenie o úspešnom vytvorení a znovu spustí akciu na načítanie kategórií, pre aktualizáciu zoznamu. Komponent slúži ako kontajner pre všetky komponenty spojené so správou kategórií.

5.2.2 Generovanie PDF s QR kódmi

PDF sa generuje na strane klienta a rovnako aj QR kódy. Tie sú najprv vygenerované pomocou knižnice `qrcode.react` a následne je vygenerovaný element canvas. Na generáciu elementu canvas z html kódu je použitá knižnica `html2canvas`. Canvas vieme funkciou `toDataURL` previesť na formát `base64`, ktorý vložíme do objektu `jsPDF` (vygenerovaný knižnicou `jsPDF`) ako obrázok s definovanou pozíciou.

Kapitola 6

Záver

Na základe požiadaviek získaných zo zadania práce, bola vytvorená webová aplikácia na evidenciu skladových zásob. Dôraz bol kladený najmä na ovládanie aplikácie zo smartfónu.

Počas tvorby aplikácie bola dôkladne preštudovaná problematika vývoja webových aplikácií zo strany servera, ale hlavne z klientskej časti. Vo veľkej miere bola aplikácia inšpirovaná aplikáciou `OfficeSpace`, ktorá svojim zameraním slúži na odlišnú vec, ale vo všeobecnosti je jej uživateľské rozhranie veľmi dobre aplikovateľné aj na aplikáciu pre evidenciu skladových zásob. Funkcionalita aplikácie splňuje požiadavky vyplývajúce zo zadania.

V prípade využitia aplikácie pre širšiu verejnosť, by bolo vhodné vytvoriť komplexjšiu správu uživateľských prístupov. Možno by v takomto prípade bolo vhodné prehodnotiť aj role jednotlivých používateľov (napríklad účty určené iba na čítanie, bez možnosti vytvárania zmien).

Pri práci som získal skúsenosti s JavaScriptovými knižnicami a frameworkami, s ktorými som doposiaľ nemal možnosť pracovať. Keby som na začiatku poznal, ako funguje knižnica na správu stavov Redux, navrhol by som klientskú časť od základov pomocou nej.

Literatúra

[1] Bhimani, K.: *Angular V/S Vue V/S React – Choose the Best in 2018*. [Online; navštíveno 01.03.2019].
URL https://www.azilen.com/blog/angular-vs-vue-vs-react-best-javascript-framework-in-2018/

[2] Bryan, P. C.; Nottingham, M.: *JavaScript Object Notation (JSON) Patch*. Duben 2013, [Online; navštíveno 20.03.2019].
URL https://tools.ietf.org/html/rfc6902

[3] Bugl, D.: *Learning Redux: Write maintainable, consistent, and easy-to-test web applications*. Packt Publishing, 2017, ISBN 978-1786462398.

[4] Deveria, A.: *Can I use getUserMedia?* [Online; navštíveno 09.03.2019].
URL https://caniuse.com/#search=getUserMedia

[5] Dresher, T.; Zuker, A.; Friedman, S.: *Hands-On Full-Stack Web Development with ASP.NET Core: Learn end-to-end web development with leading frontend frameworks, such as Angular, React, and Vue*. Packt Publishing, 2018, ISBN 978-1788622882.

[6] Hoffman, P.; Snell, J. M.: *JSON Merge Patch*. Duben 2013, [Online; navštíveno 20.03.2019].
URL https://tools.ietf.org/html/rfc7386

[7] Humphries, J.; Konsumer, D.; Muto, D.; aj.: *Practical gRPC*. Bleeding Edge Press, 2018, ISBN 9781939902580.

[8] Jones, M. B.; Bradley, J.; Sakimura, N.: *JSON Web Token (JWT)*. Květen 2015, [Online; navštíveno 01.04.2019].
URL https://tools.ietf.org/html/rfc7519

[9] Nelson, B.: *Getting to Know Vue.js*. Apress, 2018, ISBN 978-1-4842-3781-6.

[10] Porcello, E.; Banks, A.: *Learning React: Functional Web Development with React and Redux*. O'Reilly Media, 2017, ISBN 978-1491954621.

[11] Porcello, E.; Banks, A.: *Learning GraphQL: Declarative Data Fetching for Modern Web Apps*. O'Reilly Media, 2018, ISBN 978-1-492-03071-3.

[12] Salehi, S.: *Mastering Symfony*. Packt Publishing Ltd., 2016, ISBN 978-1-78439-031-0.

[13] Stauffer, M.: *Laravel Up Running*. O'Reilly Media, 2019, ISBN 978-1-492-04121-4.

[14] Subramanian, H.; Raj, P.: *Hands-On RESTful API Design Patterns Best Practices*. Packt Publishing Ltd., 2019, ISBN 978-1-78899-266-4.

[15] Votruba, T.: *PHP Framework Trends*. [Online; navštíveno 01.02.2019]. URL https://www.tomasvotruba.cz/php-framework-trends/

Príloha A

Obsah priloženého CD

- **./doc** – priečinok so súbormi dokumentácie
- **./src** – priečinok so zdrojovými súbormi webovej aplikácie
- **./src/readme.txt** – návod na spustenie